National Aeronautics and Space Administration

Shared-Memory Parallelism and OpenMP

NAS Webinar

August 22, 2012

NASA Advanced Supercomputing Division

Outline

- Shared-memory parallelism
- What is OpenMP?
- OpenMP major components
  - Fork-join execution model
  - Worksharing constructs
  - Data sharing
  - Synchronization primitives
- Use of OpenMP
- Hybrid MPI + OpenMP model
- OpenMP tasking
- Performance considerations
- Future OpenMP extensions

Topics for the next webinar

Question? Use the Webex chat facility to ask the Host

Shared-Memory Parallelism

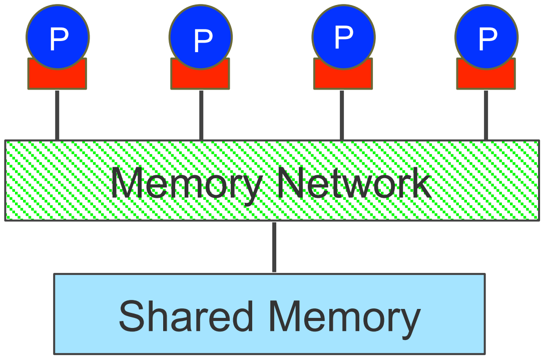
- Many modern parallel computers
  - A cluster of shared-memory nodes with multicore CPUs
- Size of shared-memory nodes getting larger
  - Increased number of cores (4, 8, 16 …)
  - Many cores in new types of systems (such as GPUs, Intel MIC)
- Shared-memory programming
  - Access to the same, globally shared, address space
  - No need for explicit data communication
  - Possibility for maintaining sequential equivalency

NASA's Pleiades Supercomputer

Question? Use the Webex chat facility to ask the Host

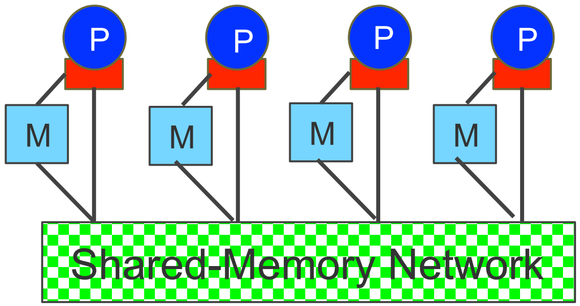
Shared-Memory Architecture

- Multiple processing units accessing global shared memory using a single address space



UMA: Uniform Memory Access

– "equidistant" access from all processors

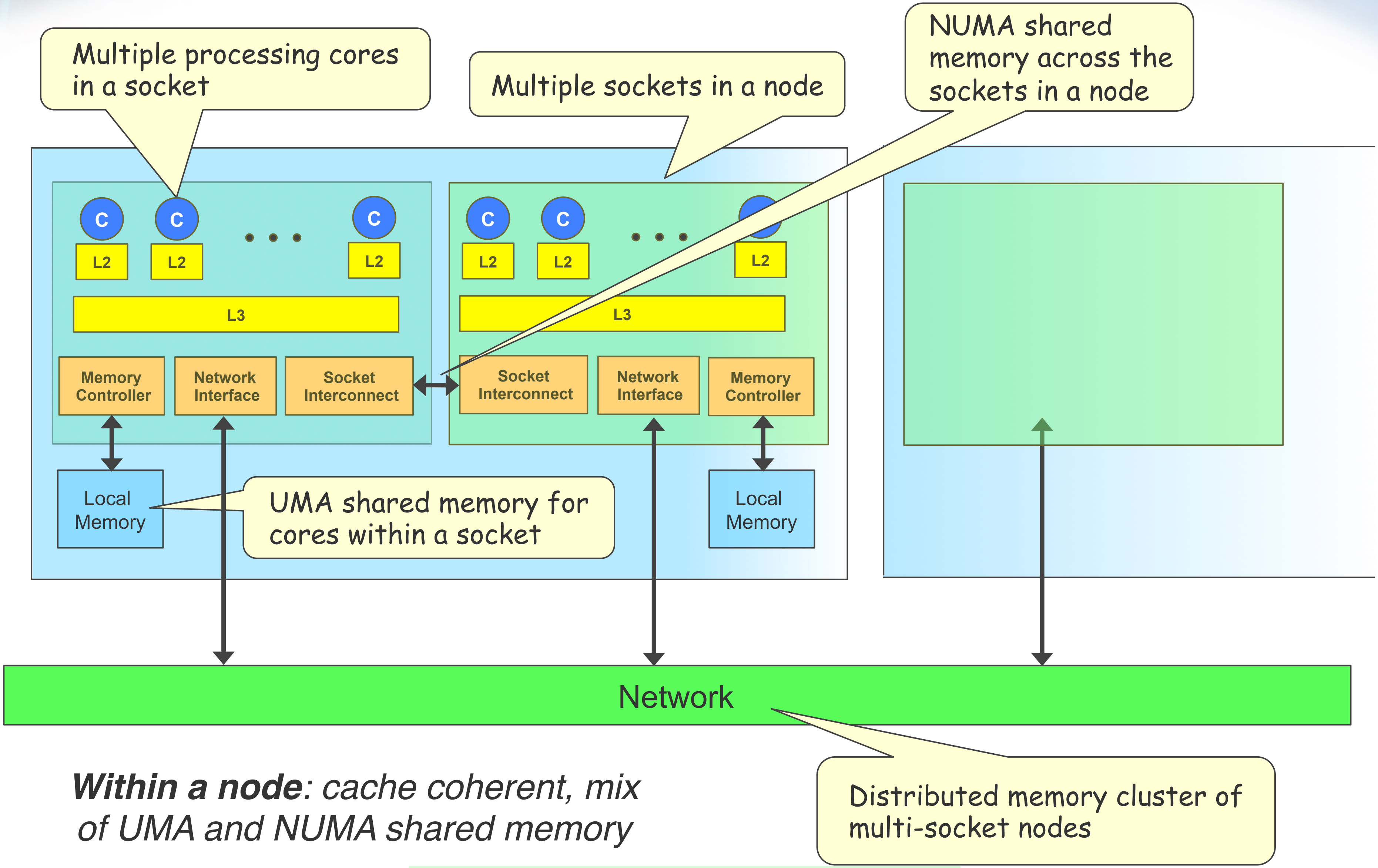


NUMA: Non-Uniform Memory Access

– local memory versus remote memory

- Shared-memory systems easier to program
  - User responsible for synchronization of processors for correct data access and modification
- Scaling to large number of processors can be an issue

A Typical Supercomputer



***Within a node**: cache coherent, mix of UMA and NUMA shared memory*

Question? Use the Webex chat facility to ask the Host

Programming Approaches

- Thread-based approaches
  - Posix threads (low level)
  - OpenMP (de-facto standard)
  - Intel Thread Building Block
- Task-based approaches
  - Intel Cilk++
  - OpenMP 3.0
  - Grand Central Dispatch from Apple
- Others
  - Global arrays
  - Compiler auto-parallelization

Question? Use the Webex chat facility to ask the Host

What is OpenMP?

- A standard API to support shared-memory multiprocessing programming
  - Compiler directives and library routines for C/C++ and Fortran
  - Specification defined and maintained by the OpenMP Architecture Review Board
    - OpenMP 1.0 released in October 1997 for Fortran, 1998 for C/C++
    - Latest 3.1 released in July 2011
  - Implemented and supported by many compiler vendors
    - (Intel, PGI, IBM, Oracle, GCC, etc.)

OpenMP®

www.openmp.org

Question? Use the Webex chat facility to ask the Host

Compiler Directives

- Special *#pragma* in C/C++, special *comments* in Fortran
- Often only enabled by a special compiler flag
- Program may be run sequentially when directives are ignored

- Examples of compiler directives

C/C++

```
#pragma omp parallel for
  for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
```

Fortran

```
!$omp parallel do
  do i = 1, n
    a(i) = b(i) + c(i)
  end do
!$omp end parallel do
```

Compile with "`-openmp`" (Intel compilers) to enable the compiler directives, otherwise they are treated as comments and the loop is run sequentially.

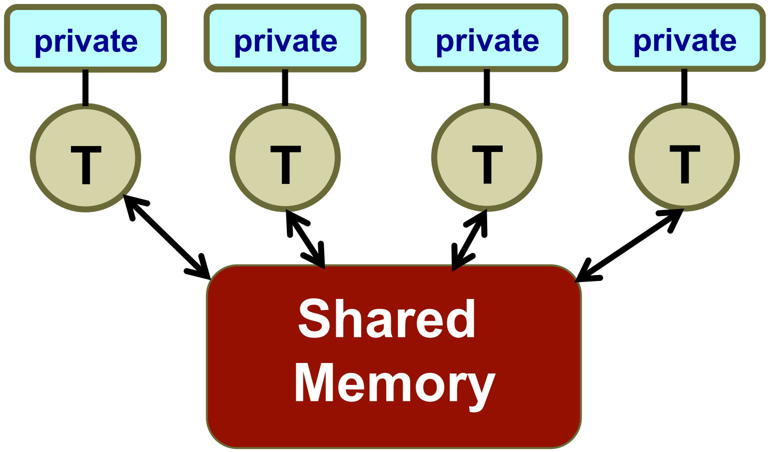
Question? Use the Webex chat facility to ask the Host

Advantages of OpenMP

- Directive-based approach
  - Possible to write sequentially consistent code
  - Easier maintenance
- Global view of application memory space
  - Relatively faster program development
- Incremental parallelization
  - Piecemeal code development
  - Easier to program and debug
- When mixed with MPI
  - Maps well with multicore hybrid architectures

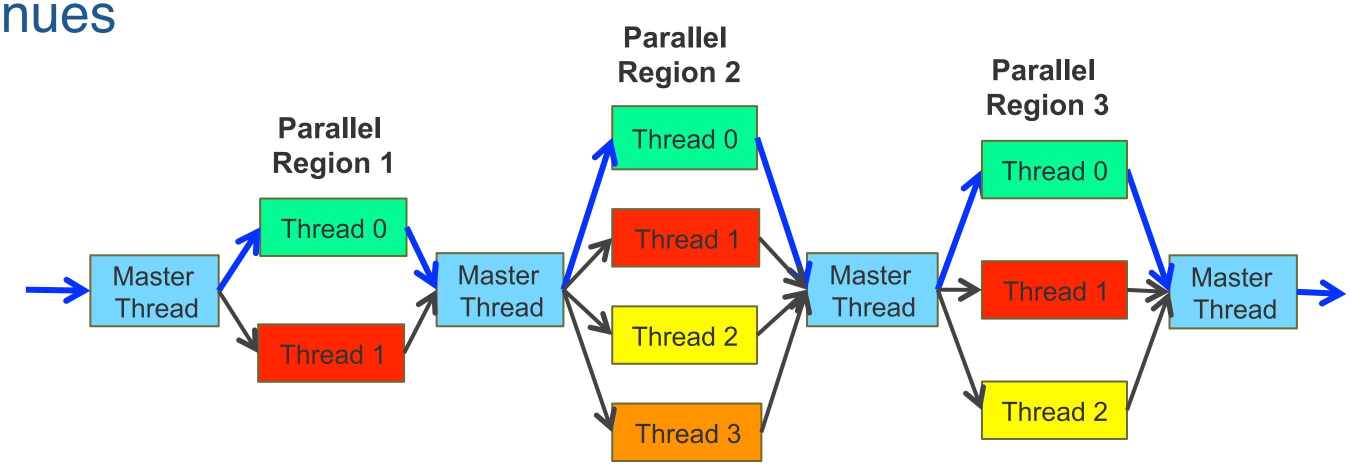
Major Components

- OpenMP thread
  - Execution entity with a stack and its private memory
  - Dynamically created and managed by the OpenMP runtime library
  - Access to shared memory
- Language components
  - *Fork-join* model for structured programming
  - Worksharing constructs for work distribution
  - Data sharing attributes
  - Synchronization primitives
- Runtime library routines
- Environment variables



Execution Model

- *Fork-join* model
  - Program starts with a single (*master*) thread
  - Multiple threads are forked by the *master* thread at a **parallel** construct
    - The *master* thread is part of the new team of threads
  - Threads perform work in the **parallel** region
    - *Worksharing* constructs distribute work among threads
    - Threads may be synchronized with synchronization constructs
  - Threads join at the end of the **parallel** region and the *master* thread continues



Parallel Construct

- The fundamental construct to start parallel execution
  - Invocation of a team of threads
  - Code executed redundantly by every thread until a *worksharing* construct is encountered
  - Number of threads controlled via
    - The **OMP_NUM_THREADS** environment variable
    - A call to **omp_set_num_threads**(), or
    - The **num_threads** clause

```
omp_set_num_threads(4);
#pragma omp parallel private(myid)
{
   myid = omp_get_thread_num();
   printf("myid is %d\n", myid);
}
```


Question? Use the Webex chat facility to ask the Host

Worksharing Construct

- The construct to distribute work among threads
  - **for** (or **do**): used to split up loop iterations among the threads

```
#pragma omp for
  for (i=0; i<n; i++) a[i] = b[i] + c[i];
```

  - **sections**: assigning consecutive but independent code blocks to different threads (can be used to specify task parallelism)
    - Each code block is indicated by the **section** directive

```
#pragma omp sections
{ #pragma omp section
  work1();
  #pragma omp section
  work2();
}
```

Worksharing Construct (cont.)

- **single**: specifying a code block executed by only one thread

```
#pragma omp single
  s = 0;
```

- There is an *implicit barrier* at the end of a worksharing construct
  - But can be suppressed with the "**nowait**" clause

```
#pragma omp for nowait
  for (i=0; i<n; i++) a[i] = b[i] + c[i];
#pragma omp for
  for (i=0; i<n; i++) d[i] = e[i] + f[i];
```

"**nowait**" suppresses the barrier at the end of the first `for` loop

- **Master** construct
  - code block executed by the master thread only, no barrier wait

Loop Scheduling

- Clause to define how loop iterations are distributed among threads of the team

```
#pragma omp for schedule(static)
  for (i=0; i<n; i++) a[i] = b[i] + c[i];
```

- Loop scheduling kinds
  - ***static***: for balanced workload, lowest overhead
    - Default for most compilers
  - ***dynamic***: for unbalanced loop iterations
  - ***guided***: for special monotonically increasing or decreasing workload
  - ***auto***: compiler determines at runtime

Question? Use the Webex chat facility to ask the Host

Data Sharing

- Accessibility of variables by threads
  - ***shared***: variable is shared by all threads in a team
  - ***private***: variable is private to each thread
  - By default, variables are shared
    - With some exceptions, such as, loop variable is private
- Specifying data sharing attribute in a parallel region or worksharing construct
  - ***shared*** clause: for variables shared by threads
  - ***private*** clause: for variables private to each thread
  - ***reduction*** clause: combining private copies of a variable to the shared copy by an operator. Reduced final value is only guaranteed at a barrier.

Question? Use the Webex chat facility to ask the Host

Data Sharing (cont.)

- An example

```
s = 0.0;
#pragma omp parallel for private(i,b) \
     shared(a) reduction(+:s)
  for (i = 0; i < n; i++) {
     b = a[i] * a[i];
     s += b;
  }
printf("sum = %g\n", s);
```

"+" is a reduction operator

- ***Threadprivate*** directive
  - Special storage for global variables, private to each thread
  - Specified at the variable declaration, valid throughout the program

```
static double c1, c2;
#pragma omp threadprivate(c1,c2)
```

Synchronization

- *Barrier:* wait until all of threads of a team have reached this point before continuing
  - **Barrier** construct specifies an *explicit* barrier
  - A worksharing construct has an *implicit* barrier at the end
- **Critical** construct
  - Code block executed by only one thread at a time, e.g., allows multiple threads to update shared data

```
#pragma omp critical
{
  s = s + s_local;
}
#pragma omp barrier
printf("sum = %g\n", s);
```

Ensure one thread updates the shared variable "**s**" at a time

All threads have done the update before the result is printed

Synchronization (cont.)

- **Atomic** construct
  - Update a shared variable atomically, can be more efficient than the **critical** construct if there is hardware support
  - Only valid for scalar variable and a limited set of operations (**+**,*****,**–**,…)

```
#pragma omp atomic
  s = s + s_local;
#pragma omp barrier
printf("sum = %g\n", s);
```

  - Other forms are also available:
    - "**atomic read**", "**atomic write**", "**atomic capture**"
- Locks
  - Similarly to **critical** but provided by the library routines and more flexible

Code Example: Sum of Squares

```
#include <omp.h>
long int sum = 0, loc_sum;
int thread_id;
#pragma omp parallel private(thread_id, loc_sum)
{
   loc_sum = 0;
   thread_id = omp_get_thread_num();
   #pragma omp for schedule(static)
   for(i = 0; i < N; i++)
   {
       loc_sum = loc_sum + i * i;
   }
   printf("Thread %d: loc_sum = %ld\n", thread_id, loc_sum);
   #pragma omp critical
   sum = sum + loc_sum;
}
printf("sum = %ld\n",sum);
```

Needed for runtime routines

Forks off the threads and starts the parallel execution; declares *thread_id* and *loc_sum* private

Each thread retrieves its own id

Worksharing construct distributes the work

Each thread prints its id and local sum

Threads cooperate to update the shared variable one by one

Master thread prints result

Use of OpenMP on Pleiades

- Basic steps
  - Select a compiler:

    ```
    module load comp-intel/11.1.072
    ```
  - Compile codes with flags that enable OpenMP

    ```
    icc -o s1.x -O3 -openmp squares.c
    ```
  - Set the number of threads to be used

    ```
    setenv OMP_NUM_THREADS 8
    ```
  - Run the executable

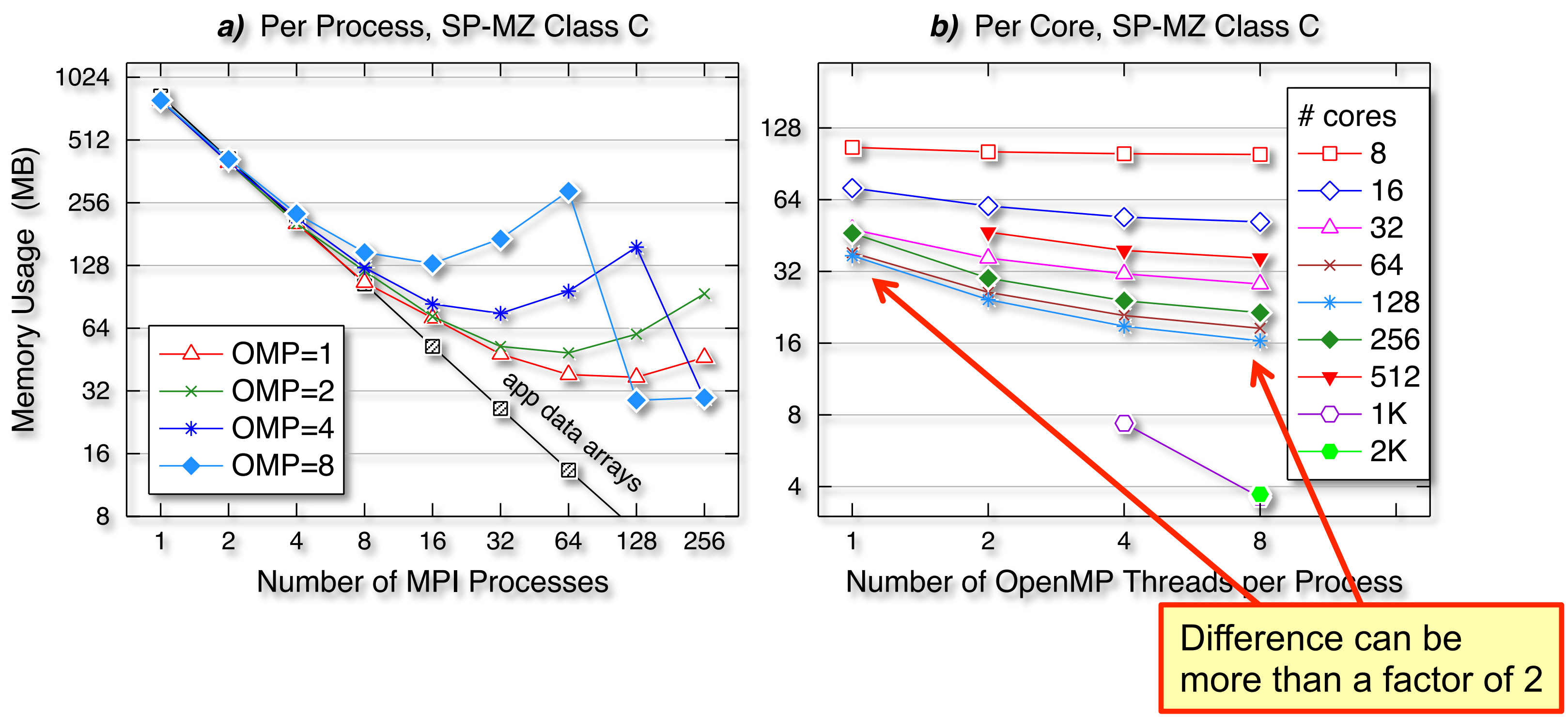
    ```
    ./s1.x
    ```
- For details see

  http://www.nas.nasa.gov/hecc/support/kb/With-OpenMP_103.html

Hybrid MPI + OpenMP

- The hybrid model
  - OpenMP works in the memory space of each MPI process
    - Shared memory within each MPI process but distributed memory across MPI processes
- Advantages of the hybrid model
  - Maps well to many hardware architectures, including Pleiades
    - MPI for communication between distributed-memory nodes
    - OpenMP for shared-memory parallelism with a node
  - Can achieve good scalability when not possible with pure MPI
  - A hybrid code may consume less memory than a pure MPI code

Memory Usage of Hybrid Codes



The SP-MZ benchmark on the SGI Altix ICE

Hybrid Programming

- Approaches
  - Common approach
    - MPI for parallelism at coarser level, OpenMP at finer level
    - No MPI calls inside OpenMP parallel regions
  - Mixed approach
    - MPI routines may be called inside OpenMP parallel regions
    - Requires the MPI library to be thread-safe (MPI_THREAD_MULTIPLE)
- Program development
  - MPI and OpenMP can be developed separately

Hybrid Code Example: Computing Pi

```
include "mpif.h"
integer myid,numprocs,ierr,n,i
real(8) h,sum,mypi,pi
call MPI_Comm_rank(MPI_COMM_WORLD,myid,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,numprocs,ierr)
n = 100000
h = 1.0d0/n
sum = 0.0d0
!$omp parallel do private(x) reduction(+:sum)
do i = myid+1, n, numprocs
   x = h * (i - 0.5d0)
   sum = sum + 4.0d0 / (1.0d0 + x * x)
end do
mypi = h * sum
call MPI_Reduce(mypi,pi,1,MPI_REAL8,MPI_SUM,    &
                0,MPI_COMM_WORLD,ierr)
if (myid.eq.0) print *,"pi is ",pi
```

Get rank and size of MPI processes

Use OpenMP to compute partial sum

Reduce the final result from all MPI processes

Rank 0 prints result

Question? Use the Webex chat facility to ask the Host

Use of MPI + OpenMP on Pleiades

- Basic steps
  - Select a compiler and an MPI library:

    ```
    module load comp-intel/11.1.072 mpi-sgi/mpt.2.06r6
    ```
  - Compile codes with flags that enable OpenMP and link with MPI library

    ```
    ifort -o s2.x -O3 -openmp pi_hybrid.f90 -lmpi
    ```
  - Set thread and process binding flags (for performance reason)

    ```
    setenv MPI_DSM_DISTRIBUTE
    setenv MPI_OPENMP_INTEROP
    ```
  - Set the number of threads to be used

    ```
    setenv OMP_NUM_THREADS 4
    ```
  - Run the executable (with 2 MPI processes, 4 OpenMP threads/process)

    ```
    mpiexec -np 2 ./s2.x
    ```
- For details see

  http://www.nas.nasa.gov/hecc/support/kb/52/

OpenMP Performance Issues

- Why is my OpenMP code not scaling? Possible issues:
  - Overhead of OpenMP constructs
  - Granularity of work units
  - Remote data access and NUMA effect
  - Load imbalance
  - False sharing of cache
  - Poor resource utilization

- We will discuss these issues and possible solutions together with other advanced OpenMP topics in the next webinar

Question? Use the Webex chat facility to ask the Host

References

- OpenMP specifications
  - www.openmp.org/wp/openmp-specifications/
- Resources
  - www.openmp.org/wp/resources/
  - www.compunity.org/
- Benchmarks
  - OpenMP Microbenchmarks from EPCC (www.epcc.ed.ac.uk/research/openmpbench)
  - NAS Parallel Benchmarks (www.nas.nasa.gov/publications/npb.html)
- Porting applications to Pleiades
  - www.nas.nasa.gov/hecc/support/kb/52/
  - www.nas.nasa.gov/hecc/support/kb/60/

Question? Use the Webex chat facility to ask the Host